Formato dei dati

- La comunicazione deve tener conto della diversa rappresentazione dei dati
  - Rappresentazione in Big Endian e Little Endian
  - Soluzione usata dai socket: network byte order (Big Endian)
- Due soluzioni per il formato dei dati
  - Soluzione usata dai socket: si trasmettono soltanto sequenze di caratteri
    - Per trasmettere un valore numerico, lo si converte in una sequenza di caratteri e poi si invia la stringa
    - Il peer conosce il tipo di dati che deve aspettarsi e lo converte nel formato opportuno (ad es. sscanf() per convertire da stringa in valore numerico)
  - In alternativa, si definisce una rappresentazione standard dei dati
    - Ad esempio, XML per SOAP

getsockname() e getpeername()

```
int getsockname(int sockfd, struct sockaddr *name, socklen_t *namelen);
int getpeername(int sockfd, struct sockaddr *nameddr, socklen_t *namelen);
```

- Forniscono indirizzo IP/porta associati ad un socket
  - getsockname: indirizzo IP/porta locali associati al socket (semi-associazione locale)
  - getpeername: indirizzo IP/porta remoti associati al peer (semi-associazione remota)
- getsockname utilizzata dal client per conoscere l'indirizzo IP ed il numero di porta locali assegnati dal SO
- Utilizzate dal server per:
  - Sapere l'indirizzo IP su cui ha ricevuto la richiesta (dopo accept())
  - Sapere la porta del socket di connessione
  - Per conoscere l'indirizzo IP e la porta del client, il server usa accept()
    - Ricordarsi di definire la dimensione della struttura sockaddr prima di invocare accept()
    - Vedere esempio successivo echo_server.c

Esempio getsockname() - server

```
…
/* dopo la funzione listen() */
/* Indirizzo IP e numero di porta assegnati al socket di ascolto */
servaddr_len = sizeof(servaddr);
getsockname(listensd, (struct sockaddr *)&servaddr, &servaddr_len);
printf("Socket di ascolto: indirizzo IP %s, porta %d\n",
        (char *)inet_ntoa(servaddr.sin_addr, ntohs(servaddr.sin_port));

for ( ; ; ) {
  cliaddr_len = sizeof(cliaddr);
  if ((connsd = accept(listensd, (struct sockaddr *)&cliaddr, &cliaddr_len)) < 0) {
    perror("errore in accept");
    exit(1);
  }
  /* Indirizzo IP e numero di porta assegnati al socket di connessione */
  getsockname(connsd, (struct sockaddr *) &servaddr, &servaddr_len);
```


Esempio getsockname() – server (2)

```
printf("Socket di connessione: indirizzo IP %s, porta %d\n",
        (char *)inet_ntoa(servaddr.sin_addr), ntohs(servaddr.sin_port));
/* Indirizzo IP e numero di porta del client: non serve chiamare getpeername */
printf("Indirizzo del client: indirizzo IP  %s, porta %d\n",
        inet_ntoa(&cliaddr.sin_addr, buff, sizeof(buff)), ntohs(cliaddr.sin_port));
if( (pid = fork()) == 0 ) {
….
```

Esempio getsockname() - client

```
…
if (connect(sockd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
  perror("errore in connect");
  exit(1);
}
/* Indirizzo IP e numero di porta assegnati dal S.O. alla connessione con
   endpoint identificato da sockd */
 localaddr_len = sizeof(localaddr);
 getsockname(sockd, (struct sockaddr *) &localaddr, &localaddr_len);
 printf("Indirizzo locale: indirizzo IP %s, porta %d\n",
         (char *)inet_ntoa(localaddr.sin_addr), ntohs(localaddr.sin_port));

 /* Indirizzo IP e numero di porta del peer; l'altro endpoint è identificato dal
   socket di connessione del server e non dal socket di ascolto.  */
 peeraddr_len = sizeof(peeraddr);
 getpeername(sockd, (struct sockaddr *) &peeraddr, &peeraddr_len);
 printf("Indirizzo del peer: indirizzo IP %s, porta %d\n",
         (char *)inet_ntoa(peeraddr.sin_addr), ntohs(peeraddr.sin_port));
….
```


Le opzioni dei socket

- L'API socket mette a disposizioni due funzioni per gestire il comportamento dei socket

```
int setsockopt(int sockfd, int level, int optname, const void *optval,
socklen_t optlen);

int getsockopt(int sockfd, int level, int optname, void *optval,
socklen_t *optlen);
```

- setsockopt() per impostare le caratteristiche del socket
- getsockopt() per conoscere le caratteristiche impostate del socket
- Entrambe le funzioni restituiscono 0 in caso di successo, -1 in caso di fallimento

Funzione setsockopt()

- Parametri della funzione setsockopt()
  - sockfd: descrittore del socket a cui si fa riferimento
  - level: livello del protocollo (trasporto, rete, …)
    - SOL_SOCKET per opzioni generiche del socket
    - SOL_TCP per i socket che usano TCP
  - optname: su quale delle opzioni definite dal protocollo si vuole operare (il nome dell'opzione)
  - optval: puntatore ad un'area di memoria contenente i dati che specificano il valore dell'opzione da impostare per il socket a cui si fa riferimento
  - optlen: dimensione (in byte) dei dati puntati da optval

Alcune opzioni generiche

- Analizziamo alcune opzioni generiche da usare come valore per optname:
  - SO_KEEPALIVE: per controllare l'attività della connessione (in particolare per verificare la persistenza della connessione)
    - optval è un intero usato come valore logico
  - SO_RCVTIMEO: per impostare un timeout in ricezione (sulle operazioni di lettura di un socket)
    - optval è una struttura di tipo timeval contenente il timeout
    - utile anche per impostare un tempo massimo per la connect()
  - SO_SNDTIMEO: per impostare un timeout in trasmissione (sulle operazioni di scrittura di un socket)
    - optval è una struttura di tipo timeval contenente il timeout
  - SO_REUSEADDR: per riutilizzare un indirizzo locale; modifica il comportamento della funzione bind() che fallisce nel caso in cui l'indirizzo locale sia già in uso da parte di un altro socket
    - optval è un intero usato come valore logico

Esempio opzione SO_REUSEADDR

```
…
int reuse = 1;
if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("errore creazione socket");
    exit(1);
}
if (setsockopt(listensd, SOL_SOCKET, SO_REUSEADDR, &reuse,
                    sizeof(int)) < 0) {
    perror("errore setsockopt");
    exit(1);
}
if (bind(listensd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
    perror("errore bind");
    exit(1);
} …
```


Server TCP iterativo (o sequenziale)

- Gestisce una connessione alla volta
  - Mentre è impegnato a gestire la connessione con un determinato client, possono arrivare altre richieste di connessione
  - Il SO stabilisce le connessioni con i client; tuttavia queste rimangono in attesa di servizio nella coda di backlog finché il server non è libero
- Più semplice da progettare, implementare e mantenere (e meno diffuso)
- Adatto in situazioni in cui:
  - il numero di client da gestire è limitato
  - il tempo di servizio per un singolo client è limitato (vedi esempio daytime)

Struttura di un server TCP iterativo

```
int listensd, connsd;

listensd = socket(AF_INET, SOCK_STREAM, 0);
bind(listensd, …);
listen(listensd, …);
for (; ;) {
        connsd = accept(listensd, …);
        do_it(connsd);   /* serve la richiesta */
        close(connsd);   /* chiude il socket di connessione */
}
```


Esempio: contatore accessi

• Esempio di applicazione client/server orientata alla connessione
  - il server conta il numero di client che accedono al suo servizio
  - il client contatta il server per conoscere tale numero
  - messaggio ASCII stampabile
  - esecuzione iterativa del server

count_client

apre la connessione con il server
ripete finché *end-of-file*:
    ricevi testo
    stampa caratteri ricevuti
chiude la connessione
esce

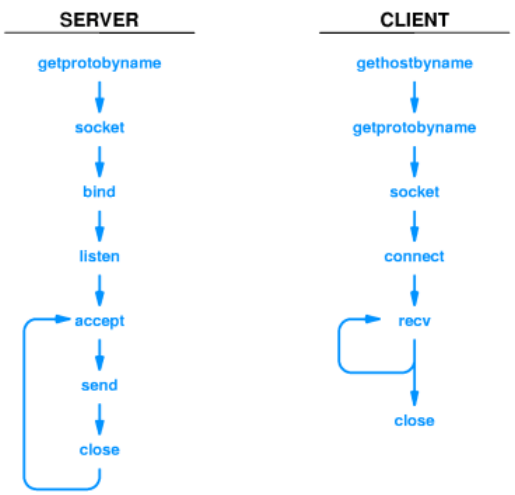
count_server

crea socket d'ascolto e si pone in attesa
ripete *forever*:
    accetta nuova connessione, usa socket di connessione
    incrementa il contatore ed invia il messaggio
    chiude il socket di connessione

Esempio: contatore accessi (2)

Uso delle funzioni nell'esempio:

- Il client chiude il socket dopo l'uso
- Il server non chiude mai il socket d'ascolto; chiude il socket di connessione dopo aver risposto al client



Client TCP count

```
/* countTCP_client.c - code for example client program that uses TCP */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#define PROTOPORT       5193          /* default protocol port number */

/* Syntax:    count_client [ host [port] ]
              host  - name of a computer on which server is executing
              port  - protocol port number server is using */
```


Client TCP count (2)

```
int main(int argc, char **argv)
{
  struct   hostent  *ptrh;              /* pointer to a host table entry */
  struct   protoent *ptrp;              /* pointer to a protocol table entry  */
  struct   sockaddr_in sad;             /* structure to hold an IP address  */
  int      sd;                          /* socket descriptor */
  int      port;                        /* protocol port number */
  char    *host;                        /* pointer to host name */
  int      n;                           /* number of characters read */
  char     buf[1000];                   /* buffer for data from the server */
  char     localhost[] = "localhost";   /* default host name */

  memset((void *)&sad, 0, sizeof(sad));       /* clear sockaddr structure */
  sad.sin_family = AF_INET;                   /* set family to Internet */

  /* Check command-line argument for protocol port and extract
     port number if one is specified.  Otherwise, use the default
     port value given by constant PROTOPORT */

  if (argc > 2) port = atoi(argv[2]);     /* if protocol port specified convert to binary */
  else port = PROTOPORT;                  /* use default port number */
```


Client TCP count (3)

```
  if (port > 0)                           /* test for legal value */
    sad.sin_port = htons(port);
  else {                                  /* print error message and exit */
    fprintf(stderr,"bad port number %s\n",argv[2]);
    exit(1); }

  /* Check host argument and assign host name. */
  if (argc > 1) host = argv[1];           /* if host argument specified  */
  else host = localhost;

  /* Convert host name to equivalent IP address and copy to sad. */
  ptrh = gethostbyname(host);
  if (ptrh == NULL ) {
    herror("gethostbyname");
    exit(1); }
  memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

/* Map TCP transport protocol name to protocol number. */
  if ( (ptrp = getprotobyname("tcp")) == NULL) {
    fprintf(stderr, "cannot map \"tcp\" to protocol number");
    exit(1);  }
```

Client TCP count (4)

```
  /* Create a socket. */
  if ((sd = socket(AF_INET, SOCK_STREAM, ptrp->p_proto)) < 0) {
    perror("socket creation failed");
    exit(1); }

  /* Connect the socket to the specified server. */
  if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
    perror("connect failed");
    exit(1);  }

/* Repeatedly read data from socket and write to user's screen. */
  n = recv(sd, buf, sizeof(buf), 0);
  while (n > 0) {
    write(1, buf, n);
    n = recv(sd, buf, sizeof(buf), 0);
  }

  close(sd);
  exit(0);
}
```


Server TCP count

```
/* countTCP_server.c - code for example server program that uses TCP */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define PROTOPORT       5193      /* default protocol port number */
#define BACKLOG         10        /* size of request queue */

int     visits      =   0;        /* counts client connections */

/* Syntax:    count_server [ port ]
              port  - protocol port number to use */
```


Server TCP count (2)

```
int main(int argc, char **argv)
{
  struct    protoent *ptrp;     /* pointer to a protocol table entry */
  struct    sockaddr_in sad;    /* structure to hold server's address */
  struct    sockaddr_in cad;    /* structure to hold client's address */
  int       listensd, connsd;   /* socket descriptors */
  int       port;               /* protocol port number */
  int       alen;               /* length of address */
  char      buf[1000];          /* buffer for string the server sends */

  memset((void *)&sad, 0, sizeof(sad)); /* clear sad */
  sad.sin_family = AF_INET;             /* set family to Internet */
  sad.sin_addr.s_addr = htonl(INADDR_ANY); /* set the local IP address */

  /* Check command-line argument for protocol port and extract
     port number if one is specified.  Otherwise, use the default
     port value given by constant PROTOPORT. */

  if (argc > 1) port = atoi(argv[1]);     /* if argument specified convert to binary */
  else port = PROTOPORT;                  /* use default port number    */
```


Server TCP count (3)

```
if (port > 0) sad.sin_port = htons(port);         /* test for illegal value */
else {                                  /* print error message and exit */
  fprintf(stderr,"bad port number %s\n",argv[1]);
  exit(1); }

/* Map TCP transport protocol name to protocol number */
if ( (ptrp = getprotobyname("tcp")) == NULL) {
   fprintf(stderr, "cannot map \"tcp\" to protocol number");
   exit(1); }

/* Create a socket */
if ((listensd = socket(AF_INET, SOCK_STREAM, ptrp->p_proto)) < 0) {
  perror("socket creation failed");
  exit(1); }

/* Bind a local address to the socket */
if (bind(listensd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
  perror("bind failed");
  exit(1); }
```

Server TCP count (4)

```
/* Specify size of request queue */
if (listen(listensd, BACKLOG) < 0) {
  perror("listen failed");
  exit(1); }

/* Main server loop - accept and handle requests */
while (1) {
  alen = sizeof(cad);
  if ( (connsd=accept(listensd, (struct sockaddr *)&cad, &alen)) < 0) {
    perror("accept failed");
    exit(1); }
  visits++;
  snprintf(buf, sizeof(buf), "This server has been contacted %d time%s\n",
                        visits, visits==1?".":"s.");
  if (write(connsd, buf, strlen(buf)) != strlen(buf)) {
    perror("error in write");
    exit(1); }

  close(connsd);
 } /* end while */
}
```


Server TCP ricorsivo (o concorrente)

- Gestisce più client (connessioni) nello stesso istante
- Utilizza una copia (processo/thread) di se stesso per gestire ogni connessione
  - Analizziamo l'uso della chiamata di sistema fork() per generare un processo figlio che eredita una connessione con un client
- I processi server padre e figlio sono eseguiti contemporaneamente sulla macchina server
  - Il processo figlio gestisce la specifica connessione con un dato client
  - Il processo padre può accettare la connessione con un altro client, assegnandola ad un altro processo figlio per la gestione
- Il numero massimo di processi figli che possono essere generati dipende dal SO

Struttura di un server TCP ricorsivo

```
int listensd, connsd;
pid_t pid;

listensd = socket(AF_INET, SOCK_STREAM, 0);
bind(listensd, …);
listen(listensd, …);
for (; ;) {
        connsd = accept(listensd, …);
        if ( (pid = fork()) == 0) {   /* processo figlio */
                close(listensd);  /* chiude il socket d'ascolto */
                do_it(connsd);    /* serve la richiesta */
                close(connsd);    /* chiude il socket di connessione */
                exit(0);          /* termina */
        }
        close(connsd);   /* il padre chiude il socket di connessione */
}
```

N.B.: nessuna delle due chiamate a close() evidenziate in rosso causa l'innesco della sequenza di chiusura della connessione TCP perché il numero di riferimenti al descrittore non si è annullato

fork()

pid_t fork(void);

- Permette di creare un nuovo processo figlio
  - È una copia esatta del processo padre
  - Eredita tutti i descrittori del processo padre
- In caso di successo restituisce un risultato sia al padre che al figlio
  - Al padre restituisce il **pid** (*process id*) del figlio
  - Al figlio restituisce 0
- Il processo figlio è una *copia* del padre
  - Riceve una copia dei segmenti testo, dati e stack
  - Esegue esattamente lo stesso codice del padre
  - La memoria è copiata (non condivisa!): quindi padre e figlio vedono valori diversi delle variabili

Server echo ricorsivo

- Echo: il server replica un messaggio inviato dal client
- Il client legge una riga di testo dallo standard input e la invia al server
- Il server legge la riga di testo dal socket e la rimanda al client
- Il client legge la riga di testo dal socket e la invia allo standard output

echo_server.c

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <strings.h>
#include <time.h>

#define SERV_PORT       5193
#define BACKLOG         10
#define MAXLINE         1024
.....
int main(int argc, char **argv)
{
```


echo_server.c (2)

```
pid_t                   pid;
int                     listensd, connsd;
struct sockaddr_in      servaddr, cliaddr;
int                     len;

if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
  perror("errore in socket");
  exit(1); }

memset((char *)&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

if ((bind(listensd, (struct sockaddr *)&servaddr, sizeof(servaddr))) < 0) {
  perror("errore in bind");
  exit(1); }
```


echo_server.c (3)

```
if (listen(listensd, BACKLOG) < 0 ) {
  perror("errore in listen");
  exit(1);
}

for ( ; ; ) {
  len = sizeof(cliaddr);
  if ((connsd = accept(listensd, (struct sockaddr *)&cliaddr, &len)) < 0) {
    perror("errore in accept");
    exit(1);
  }

  if ((pid = fork()) == 0) {
    if (close(listensd) == -1) {
      perror("errore in close");
      exit(1); }
    printf("%s:%d connesso\n", inet_ntoa(cliaddr.sin_addr),
                                        ntohs(cliaddr.sin_port));
```

echo_server.c (4)

```
      str_srv_echo(connsd);  /* svolge il lavoro del server */

      if (close(connsd) == -1) {
        perror("errore in close");
        exit(1);
      }
      exit(0);
    } /* end fork */

    if (close(connsd) == -1) {          /* processo padre */
      perror("errore in close");
      exit(1);
    }
  } /* end for */
}
```


echo_server.c (5)

```
void str_srv_echo(int sockd)
{
  int       nread;
  char      line[MAXLINE];

  for ( ; ; ) {
    if ((nread = readline(sockd, line, MAXLINE)) == 0)
                        /* readline restituisce il numero di byte letti */
      return; /* il client ha chiuso la connessione e inviato EOF */

    if (writen(sockd, line, nread)) {
      fprintf(stderr, "errore in write");
      exit(1);
    }
  }
}
```


echo_client.c

```
int main(int argc, char **argv)
{
  int                   sockfd;
  struct sockaddr_in    servaddr;

  if (argc != 2) {
    fprintf(stderr, "utilizzo: echo_client <indirizzo IP server>\n");
    exit(1);
  }
  if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("errore in socket");
    exit(1);
  }
  memset((void *)&servaddr, 0, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_port = htons(SERV_PORT);
  if (inet_pton(AF_INET, argv[1], &servaddr.sin_addr) <= 0) {
    fprintf(stderr, "errore in inet_pton per %s", argv[1]);
    exit(1);
  }
```


echo_client.c (2)

```
if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
    perror("errore in connect");
    exit(1);
  }

  str_cli_echo(stdin, sockfd);          /* svolge il lavoro del client */

  close(sockfd);

  exit(0);
}
```

echo_client.c (3)

```
void str_cli_echo(FILE *fd, int sockd)
{
  char     sendline[MAXLINE], recvline[MAXLINE];
  int      n;

  while (fgets(sendline, MAXLINE, fd) != NULL) {
    if ((n = writen(sockd, sendline, strlen(sendline))) < 0) {
      perror("errore in write");
      exit(1);
    }

    if ((n = readline(sockd, recvline, MAXLINE)) < 0) {
      fprintf(stderr, "errore in readline");
      exit(1);
    }

   fputs(recvline, stdout);
  }
}
```


Analisi applicazione echo

- Il comando **netstat** permette di ottenere informazioni sullo stato delle connessioni instaurate
  - Opzione -a: per visualizzare anche lo stato dei socket non attivi
  - Opzione -Ainet: per specificare la famiglia di indirizzi Internet
  - Opzione –n: per visualizzare gli indirizzi numerici (invece di quelli simbolici) degli host e delle porte
  - Nell'esempio applicazioni client e server eseguite sulla stessa macchina

$ netstat -a -Ainet -n

Proto	Recv-Q	Send-Q	Local Address	Foreign Address	State
tcp	0	0	127.0.0.1:5193	127.0.0.1:1232	ESTABLISHED
tcp	0	0	127.0.0.1:1232	127.0.0.1:5193	ESTABLISHED
tcp	0	0	0.0.0.0:5193	0.0.0.0:*	LISTEN

Analisi applicazione echo (2)

- Il comando **ps** (process state) permette di ottenere informazioni sullo stato dei processi
  - Opzione l: formato lungo
  - Opzione w: output largo
  - Nell'esempio applicazioni client e server eseguite sulla stessa macchina

$ ps lw

F	UID	PID	PPID	PRI	NI	VSZ	RSS	WCHAN	STAT	TTY	TIME	COMMAND
000	501	31276	31227	1	0	1080	296	wait_f	S	pts/2	0:00	echo_server
044	501	31308	31276	1	0	1084	360	tcp_re	S	pts/2	0:00	echo_server
000	501	31393	31166	9	0	1084	336	read_c	S	pts/1	0:00	echo_client 127.0.0.1

I segnali

- I segnali sono interruzioni software inviate ad un processo
- Permettono di notificare ad un processo l'occorrenza di qualche evento asincrono
  - Inviati dal kernel o da un processo
  - Usati dal kernel per notificare situazioni eccezionali (ad es. errori di accesso, eccezioni aritmetiche)
  - Usati anche per notificare eventi (ad es. terminazione di un processo figlio)
  - Usati anche come forma elementare di IPC
- Ogni segnale ha un nome, che inizia con SIG; ad es.:
  - SIGCHLD inviato dal SO al processo padre quando un processo figlio è terminato o fermato
  - SIGALRM: generato quando scade il timer impostato con la funzione alarm()
  - SIGKILL: per terminare immediatamente (kill) il processo
  - SIGSTOP: per fermare (stop) il processo
  - SIGUSR1 e SIGUSR2: a disposizione dell'utente per implementare una forma di comunicazione tra processi

Gestione dei segnali

- Un processo può decidere quali segnali gestire
  - Ovvero le notifiche di segnali che accetta
  - Per ogni segnale da gestire, deve essere definita un'apposita funzione di gestione (**signal handler**)
- Il segnale viene consegnato al processo quando viene eseguita l'azione per esso prevista
- Per il tempo che intercorre tra la generazione del segnale e la sua consegna al processo, il segnale rimane *pendente*
- Ogni segnale ha un gestore (handler) di default
  - Alcuni segnali non possono essere ignorati e vengono gestiti sempre (SIGKILL e SIGSTOP)
  - Alcuni segnali vengono ignorati per default (es. SIGCHLD)
    - Tale comportamento può tuttavia essere modificato

Gestione dei segnali (2)

- Per tutti i segnali non aventi un'azione specificata che è fissa, il processo può decidere di:
  - Ignorare il segnale
  - Catturare il segnale
  - Accettare l'azione di default propria del segnale
- La scelta riguardante la gestione del segnale può essere specificata mediante le funzioni signal() e sigaction()
- Per approfondimenti vedere GaPiL (Guida alla Programmazione in Linux)

Segnale SIGCHLD

- Quando un processo termina (evento asincrono) il kernel manda un segnale SIGCHLD al padre ed il figlio diventa zombie
  - Mantenuto dal SO per consentire al padre di controllare il valore di uscita del processo e l'utilizzo delle risorse del figlio
  - Per default, il padre ignora il segnale SIGCHLD ed il figlio rimane zombie finché il padre non termina
- Per evitare di riempire di zombie la tabella dei processi bisogna fornire un handler per SIGCHLD
- Il processo zombie viene rimosso quando il processo padre chiama le funzioni wait() o waitpid()
  - Definite in sys/wait.h
  - wait(): sospende il processo corrente finché non termina un qualunque processo figlio
  - waitpid(): non bloccante (se opzione WNOHANG); permette di specificare quale processo attendere e, chiamata all'interno di un ciclo, consente di catturare tutti i segnali

Handler per SIGCHILD

```
#include <signal.h>
#include <sys/wait.h>
void sig_chld_handler(int signum)
{
  int     status;
  pid_t   pid;

  while ((pid = waitpid(WAIT_ANY, &status, WNOHANG)) > 0)
    printf ("child %d terminato\n", pid);
  return;
}
```

pid_t waitpid(pid_t pid, int * status, int options);

waitpid ritorna 0 quando non c'è nessun figlio di cui non è stato ancora ricevuto dal padre lo stato di terminazione

- waitpid() permette di scegliere quale figlio attendere sulla base del valore dell'argomento pid
  - WAIT_ANY (oppure -1) per il primo che termina
- Quando un figlio termina e lancia il segnale SIGCHILD, waitpid() lo cattura e restituisce il pid; il processo figlio può essere rimosso

Attivazione dell'handler

```
#include <signal.h>
#include <sys/wait.h>

typedef void Sigfunc(int);
Sigfunc *signal(int signum, Sigfunc *func)
{
  struct sigaction   act, oldact;

  act.sa_handler = func;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  if (signum != SIGALRM)
     act.sa_flags |= SA_RESTART;
  if (sigaction(signum, &act, &oldact) < 0)
    return(SIG_ERR);
  return(oldact.sa_handler);
}
```

signal() attiva l'handler: prende in ingresso il numero del segnale ed il puntatore all'handler

la struttura sigaction memorizza informazioni riguardanti la manipolazione del segnale

insieme di segnali bloccati durante l'esecuzione dell'handler

la funzione sigaction() prende in ingresso una struttura con il puntatore all'handler, una maschera di segnali da mascherare e vari flag e installa l'azione per il segnale

flag SA_RESTART per far ripartire le chiamate di sistema interrotte dal segnale

echo_server con gestione SIGCHLD

```
…
 if ((listensd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
   fprintf(stderr, "errore in socket");
   exit(1); }
…
 if ((bind(listensd, (struct sockaddr *)&servaddr, sizeof(servaddr))) < 0) {
   fprintf(stderr, "errore in bind");
   exit(1); }
 if (listen(listensd, QLEN) < 0) {
   fprintf(stderr, "errore in listen");
   exit(1); }
 if (signal(SIGCHLD, sig_chld_handler) == SIG_ERR) {
   fprintf(stderr, "errore in signal");
   exit(1); }
…
```


Gestione SIGALRM

- Per evitare che un client UDP o un server UDP rimangono indefinitamente bloccati su recvfrom() si può usare il segnale di allarme SIGALRM
- E' il segnale del timer dalla funzione alarm()

  unsigned int alarm(unsigned int seconds);
  - alarm() predispone l'invio di SIGALRM dopo *seconds* secondi, calcolati sul tempo reale trascorso (il clock time)
  - Restituisce il numero di secondi rimanenti all'invio dell'allarme programmato in precedenza
  - alarm(0) per cancellare una programmazione precedente del timer

Client UDP daytime con SIGALRM

```
#define TIMEOUT            20
void sig_alrm_handler(int signo)
{
}
…
int main(int argc, char *argv[ ])
{
  …
  struct sigaction sa;
  …
  sa.sa_handler = sig_alrm_handler;       /* installa il gestore del segnale */
  sa.sa_flags = 0;
  sigemptyset(&sa.sa_mask);
  if (sigaction(SIGALRM, &sa, NULL) < 0) {
    fprintf(stderr, "errore in sigaction");
    exit(1);
  }
  if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) { /* crea il socket */
    fprintf(stderr, "errore in socket");
    exit(1);
  }
```

Client UDP daytime con SIGALRM (2)

```
…
/* Invia al server il pacchetto di richiesta*/
  if (sendto(sockfd, NULL, 0, 0, (struct sockaddr *) &servaddr,
     sizeof(servaddr)) < 0) {
    fprintf(stderr, "errore in sendto");
    exit(1);
  }
  alarm(TIMEOUT);
  n = recvfrom(sockfd, recvline, MAXLINE, 0 , NULL, NULL);
  if (n < 0) {
    if (errno != EINTR) alarm(0);
    fprintf(stderr, "errore in recvfrom");
    exit(1);
  }
  alarm(0);
…
```


Progettazione di applicazioni di rete robuste

- Nel progettare applicazioni di rete robuste si deve tener conto di varie situazioni anomale che si potrebbero verificare (la rete è inaffidabile!)
- Nell'applicazione echo due situazioni critiche:
  - Terminazione precoce della connessione effettuata dal client (invio RST) prima che il server abbia chiamato accept()
  - Terminazione precoce del server, ad esempio del processo figlio per un errore fatale: il server non ha il tempo di mandare nessun messaggio al client
- Per la soluzione delle due situazioni critiche vedi GaPiL